UNIVERSIDAD AUTÓNOMA METROPOLITANA
Unidad Iztapalapa

Práctica #9

Olvera Monroy Gonzalo

<2173011224>

Sistemas Operativos

Prof. Orlando Muñoz Texzocotetla

Trimestre: 20 - O

Desarrollo.

Implementación para imprimir una sola partición e imprimir el estado de toda la memoria.

Como había mencionado en una minipráctica en donde se pide que solo imprima una sola partición.

```
//Imprime solo una particion de la memoria
void imprimir_particion(struct Nodo *primer_nodo) {
    printf("       \t%d      \t|  %d      \t%d       \t%d    |", primer_nodo-
>contador_proceso,primer_nodo->tamanio_particion,
    primer_nodo->tamanio_ocupado, primer_nodo->tamanio_desocupado);
}
```

Así es como se imprime solo una partición, coloque unos "|" para simular la memoria.

Y el siguiente método imprime toda la lista a lo cual la llame **estado_memoria:**

```
//Imprime toda la memoria
void estado_memoria(struct Nodo *primer_nodo) {
    printf("\n\n********************* MEMORIA *********************************\n")
;
    printf("\n    Contador     Valor      Ocupados      Desocupados");
    while (primer_nodo != NULL) {
        printf("\n");
        imprimir_particion(primer_nodo);
        primer_nodo = primer_nodo->siguiente_nodo;
    }
    printf("\n");
    printf("\n*****************************************************************\n");

}
```

Así se imprime toda la memoria, el formato que elegí fue en columnas ya que se me resulto más fácil hacerlo de esa manera.

En la parte del **main** implemente un pequeño menú.

```
int main(){
    int opcion;

    // Creamos la lista ligada
    struct Nodo *primer_nodo = NULL;


    while(1){
```

```
        printf("\n\nElija un algoritmo:");
        printf("\n1. Crear Lista\n2. First-fit\n3. Best-fit\n4. Worst-fit\n5. Block-
splitting\n6. Salir");
        printf("\nIntroduce opcion: ");
        scanf("%d",&opcion);

        switch(opcion){

            case 1:
                //srand(time(0));
                primer_nodo = crea_nodo_lista_random(primer_nodo);
                estado_memoria(primer_nodo);
                break;

            case 2:
                srand(time(0));
                first_fit(primer_nodo);
                printf("\n\t\t    ---FIRST-FIT---");
                estado_memoria(primer_nodo);
                break;

            case 3:
                srand(time(0));
                best_fit(primer_nodo);
                printf("\n\t\t    ---BEST-FIT---");
                estado_memoria(primer_nodo);
                break;

            case 4:
                srand(time(0));
                worst_fit(primer_nodo);
                printf("\n\t\t    ---WORST-FIT---");
                estado_memoria(primer_nodo);
                break;

            case 5:
                srand(time(0));
                block_splitting(primer_nodo);
                printf("\n\t\t   ---BLOCK-SPLITTING---");
                estado_memoria(primer_nodo);
                break;

            case 6:
                exit(0);
```

```
            default:
                printf("\nOpcion invalida\n");
                break;
        }

    }
    return 0;
}
```

Para poder insertar los procesos siempre se tiene que crear la lista.
En donde el usuario puede elegir como quiere insertar un proceso, ya sea mediante **First-fit** que es el número 2, **Best-Fit** el número 3**, Worst-Fit** el número 4 **y Block Splitting** es el número 5.
Utilice **srand(time(0))** para que cada vez que el usuario elija un algoritmo se inserten números aleatorios diferentes y no los mismos.

Implementación para una Lista-Ligada con números aleatorios.

```
// Genera numeros aleatorios en un rango.
// Genera numeros aleatorios en un rango.
int generador_random(int l, int r) { //Este generara un numero aleatoroi en un rango l y
r
   return (rand() % (r - l + 1)) + l;
}
```

Lo primero que hice fue investigar como generar números aleatorios dando un rango en cual se van a generar. Entonces hice un método, ya que se va a ocupar para crear la lista, también para las asignaciones de memoria, que son:

1. First-Fit.
2. Best-Fit.
3. Worst-Fit.
4. Block splitting.

```
/*
Metodo que crea la lista con numeros aleatorios en un rango
Entre los numeros 5 y 10.
Y la suma de ellos deben de dar 500.
*/
struct Nodo *crea_nodo_lista_random(struct Nodo *primer_nodo){
    int total = 0, num_random, posicion=0, contador_proceso = 0;
    struct Nodo *aux;

    while (total <= TAMANIO) {
        num_random = generador_random(5, 10); // Genera los numeros aleatorios.
```

```
        primer_nodo = inserta_inicio(primer_nodo, num_random); // Se insertan los numeros
 a la lista.
        total += num_random; // Suma todos los numeros para que la suma sea 500
   }

    aux = primer_nodo;
    while(aux != NULL){
        aux->contador_proceso = 0; // Contabiliza que proceso lleva mas tiempo ocupado
        aux->posicion = posicion; // Contabiliza los nodos de la lista
        posicion++;
        aux = aux->siguiente_nodo;
    }

    printf("Total: %d", total);
    return primer_nodo;
}
```

Para la creación de la lista implemente un método llamado "**crea_nodo_lista_random**" el cual se llevó a cabo de la siguiente manera:

1. Se declararon 4 variables de tipo entero y una variable de tipo Nodo:
    - La variable **total** se encarga de sumar todos los números, para que la suma sea 500.
    - La variable **num_random** es para llamar al método **numero_aleatorio,** que va a generar los números aleatorios.
    - La variable **posición** es para saber la posición del nodo.
    - La variable **contador_proceso** es para saber cuánto tiempo lleva un proceso con un valor ocupado.
    - La variable **aux** es para estar recorriendo la lista.
2. En el primer **while** se ocupa para generar los números aleatorios, dentro del **while num_aleatorio** se le asigna el método **numero_aleatorio**, para estar insertando los valores a la lista, la variable **primer_nodo** le asigno el método de **inserta_inicio** y le paso como parámetros **primer_nodo** y **num_aleatorio**, por último, **total** suma los números para que sumen 500.
3. En el segundo **while** es para estar recorriendo la lista **aux->contador_proceso=0** contabiliza si un proceso está en la memoria para empezar en 0, **aux->posicion=posicion** es para saber la posición de cada nodo.

Implementación para First-Fit.

```
/*
Busca en la lista agujeros libres en la lista y asigna el primero en la lista que
sea lo suficientemente grande para acomodar al proceso deseado.
```

```
*/
void first_fit(struct Nodo *primer_nodo ){
    int i,tamanio_particion;

    if(esta_vacia(primer_nodo)) { // Veririfica que la lista este vacia

        printf("\nLa lista esta vacia\n");
        return;
    }

    for(i = 0; i < 100; i++) { // Se le asigna 100 veces una partición a la lista
        tamanio_particion = generador_random(3, 10); // Genera los numeros aleatorios entre 3 y 10
        while(primer_nodo != NULL){ // Recorre la lista
            if(primer_nodo->tamanio_ocupado != 0) { // Contabiliza si un proceso tiene un valor en espacio ocupado
                primer_nodo->contador_proceso ++; // Se aumenta el contador
                if(primer_nodo-
>contador_proceso  > CONTADOR){ // Si un proceso ya lleva mas de 5 unidades de tiempo se reinicia
                    primer_nodo->tamanio_ocupado = 0; // El tamanio ocupado se iniicaliza en 0
                    primer_nodo->tamanio_desocupado = primer_nodo-
>tamanio_particion; // El tamanio desocupado se reinicia
                    primer_nodo->contador_proceso = 0; // Contador se inicializa en 0
                }
            } else {
                if(tamanio_particion <= primer_nodo->tamanio_desocupado) { // Inserta el valor en el nodo
                    primer_nodo-
>tamanio_ocupado += tamanio_particion; // Se suma el valor insertado en el espacio ocupado
                    primer_nodo->tamanio_desocupado -
= tamanio_particion; // Resta el valor insertado menos el valor
                    break;
                }
            }
            primer_nodo = primer_nodo->siguiente_nodo; // Avanza al siguiente nodo
        }
    }
}
```

La implementación para **First-fit** se llevó acabo de la siguiente manera:

1. Declare 2 variables de tipo entero:
   - La variable **i** es para llevar un conteo.
   - La variable **tamanio_particion** es la que va a estar insertando el valor en la lista.
2. El **for** es para estar insertando 100 veces los valores a la lista y no estar insertando uno por uno. Dentro del **for** empiezo a generar los números aleatorios.
3. El **while** es para recorrer toda la lista, dentro de ese hay:

- Un **if(primer_nodo->tamanio_ocupado != 0)** que es para verificar si un proceso ya está en memoria, y empezar a contabilizar las 5 unidades de tiempo que debe de estar ahí. Después de eso se empieza a aumentar el contador en 1.
    - Si no es diferente de 0 entonces se hace otra verificación **if(tamanio_particion <= primer_nodo->tamanio_desocupado)** que es para empezar a insertar un proceso en la memoria. Después se le suma el valor insertado al **tamanio_ocupado** y después se le resta al **tamanio_desocupado** menos el valor insertado.
- El siguiente **if(primer_nodo->contador_proceso > CONTADOR)** es para verificar si el **tamanio_particion** ya lleva más de 5 unidades de tiempo en la memoria. NOTA: **CONTADOR** es el valor 5 que es el límite que debe de estar un proceso en memoria.
- Si el proceso es mayor a 5 entonces se libera de la memoria y se reinicia el espacio de memoria ocupado por el proceso.

Implementación para Best-Fit.

La implementación para **Best-Fit** al principio había hecho un método en donde solo se insertaba el proceso con el mismo número en la memoria es decir si el proceso era 10 se insertaba en un bloque de memoria igual al 10, pero si un proceso era igual a 3 se supone que se tenía que insertar en el 5 lo cual no lo hacía.

Entonces leyendo de nuevo las notas e investigando un poco por internet me di cuenta de que se tiene que implementar un método de búsqueda en donde se tiene que buscar la posición en donde se va a insertar el proceso y se ajuste con el valor que tiene la memoria sin dejar tanto desperdicio.

```
/*
 * Metodo que encuentra la posicion de bloque de memoria que mejor se ajuste al proceso insertado
*/
int buscar_best_nodo(struct Nodo *primer_nodo, int tamanio_particion) {
    int best_posicion = -1; // Empieza desde afuera la lista
    int tamanio_nodo = 0; // Es el tamanio que tiene el nodo

    if(esta_vacia(primer_nodo)){ //Verifica que la lista este vacia
        printf("\nNo hay nodos\n");
        return -1;
    }

    while (primer_nodo != NULL ){ //Recorre la lista
```

```
        if(primer_nodo-
>tamanio_ocupado != 0) { // Contabiliza si un proceso tiene un valor en espacio ocupado
            primer_nodo->contador_proceso++; // Se aumenta el contador
            if(primer_nodo-
>contador_proceso > CONTADOR) { // Si un proceso ya lleva mas de 5 unidades de tiempo se reinicia
                primer_nodo->tamanio_ocupado = 0; // El tamanio ocupado se iniicaliza en 0
                primer_nodo->tamanio_desocupado = primer_nodo-
>tamanio_particion; // El tamanio desocupado se reinicia al valor que tenia antes
                primer_nodo->contador_proceso = 0;  // Contador se inicializa en 0
            }
        } else {
            if (primer_nodo-
>tamanio_desocupado >= tamanio_particion) { // Verifica para poder insertar el proceso en el mejor b
loque de memoria que se ajusta
                /*
                Comience eligiendo cada proceso y encuentre el tamanio de bloque mínimo
                que se puede asignar al proceso por insertar
                */
                if(best_posicion == -1) {
                    tamanio_nodo = primer_nodo->tamanio_desocupado;
                    best_posicion = primer_nodo->posicion;
                } else {
                    /*
                    Si no, deja ese proceso y sigue revisando
                    los procesos posteriores.
                    */
                    if (tamanio_nodo > primer_nodo->tamanio_desocupado) {
                        best_posicion = primer_nodo->posicion;
                        tamanio_nodo = primer_nodo->tamanio_desocupado;
                    }

                }
            }
        }
        primer_nodo = primer_nodo->siguiente_nodo;
    }

    return best_posicion; // Se regresa la mejor posicion
}
```

Como había comentado anteriormente implemente un método llamado **buscar_best_nodo** lo cual sirve para que el proceso que se va a insertar empiece a buscar en toda lista posición por posición, hasta encontrar el valor adecuado en la lista.

1. Declare dos variables de tipo entero:

- La variable **best_posicion** se va a encargar de encontrar la mejor posición para el proceso insertado.
- La variable **tamanio_nodo** es para asignarle el **tamanio_desocupado** que tiene la memoria e ir comparando si le conviene insertar al proceso.

2. Al igual que **First-Fit** el **while** es para ir recorriendo toda la lista, dentro de ese hay:
   - Un **if(primer_nodo->tamanio_ocupado != 0)** que es para verificar si un proceso ya está en memoria, y empezar a contabilizar las 5 unidades de tiempo que debe de estar ahí. Después de eso se empieza a aumentar el contador en 1.
   - El segundo **if(primer_nodo->contador_proceso > CONTADOR)** anidado es para verificar si el **tamanio_particion** lleva más de 5 unidades de tiempo en la memoria.
   - Si no se cumple la primera condición del **if** en donde se compara si el **tamanio_ocupado** es diferente de 0, entonces hay 3 **if's** después**.**
   - El primer **if(primer_nodo>tamanio_desocupado>=tamanio_particion)** es para esta insertando en la lista el proceso.
   - El segundo **if(best_posicion == -1)** comienza a elegir cada proceso que encuentre el tamaño del bloque mínimo de memoria que se puede asignar al proceso por insertar.
     - **if(tamanio_nodo > primer_nodo->tamanio_desocupado)** Si no, deja ese proceso y sigue revisando los procesos posteriores.
   - Se regresa la mejor posición que encontró el proceso.

```
/*
Este metodo compara el tamaño de memoria requerido por el proceso con los agujeros libres
en la lista. Le asigna al proceso el agujero más pequeño que se ajuste al proceso.
*/
void best_fit(struct Nodo *primer_nodo) {
    int tamanio_particion, i;
    int best_posicion = -1;

    if(esta_vacia(primer_nodo)){ // Verifica que la lista este vacia
        printf("\nLa lista esta vacia\n");
        return;
    }

    for (i = 0; i < 100; i++){ // Se le asigna 100 veces una particion a la lista
        tamanio_particion = generador_random(3, 10); // Genera los numeros aleatorios entre 3 y 10
        best_posicion = buscar_best_nodo(primer_nodo, tamanio_particion); // Empieza la busqueda par
a inserta el proceso
        if( best_posicion != -
1) { // Si se puede encontrar un bloque de memoria para el proceso insertado
            while (primer_nodo != NULL){ // Empieza a recorrer la lista
```

```
                if(best_posicion == primer_nodo->posicion) { // Si el la posicion del bloque de memoria es igual a la posicion del nodo
                    primer_nodo->tamanio_desocupado -= tamanio_particion; // Se suma el valor insertado en el espacio ocupado
                    primer_nodo->tamanio_ocupado += tamanio_particion; // Resta el valor que tiene el tamnio desocupador menos el valor insertado
                    break;
                }
                primer_nodo = primer_nodo->siguiente_nodo; // Avanza al siguiente nodo
            }
        }

    }
}
```

Luego de hacer el método de **buscar_best_nodo** implemente el método para **best_fit.**

1. Declare tres variables de tipo entero:
    - La variable **tamanio_particion** es la que va a estar insertando el valor en la lista.
    - La variable **i** va a llevar el conteo.
    - La variable **best_posicion** es para asignarle el método **buscar_best_nodo**.
2. Se tiene el primer **if** que solo verifica si la lista no está vacía.
3. El **for** es para estar insertando 100 veces los valores a la lista y no estar insertando uno por uno. Dentro del **for** empiezo a generar los números aleatorios y le asigno a la variable **best_posicion** el método de **buscar_best_nodo** para que empiece la búsqueda.
4. Después del **for** hay un **if(best_posicion != 1)** es para ver si puede encontrar un bloque de memoria para el proceso insertado.
5. El **while** es para estar recorriendo la lista, dentro de eso hay:
    - Un **if(best_posicion == primer_nodo->posicion)** verifica que la posición que encontró el proceso sea igual a la del nodo, para poder insertar el proceso.
    - Después se le suma el valor insertado al **tamanio_ocupado** y después se le resta al **tamanio_desocupado** menos el valor insertado.

Implementación para Worst-Fit.

Al igual que **Best-Fit** se tuvo que implementar un método llamado **buscar_worst_nodo** lo cual sirve para que el proceso que se va a insertar empiece a buscar en toda lista posición por posición, hasta encontrar un valor mayor.

```c
/*
 * Metodo que encuentra la posicion de bloque de memoria que peor se ajuste al proceso insertado
*/
int buscar_worst_nodo(struct Nodo *primer_nodo, int tamanio_particion) {
    int worst_posicion = -1; // Empieza desde afuera de la lista
    int tamanio_nodo = 0; // Es el tamanio que tiene el nodo

    if(esta_vacia(primer_nodo)){ //Verifica que la lista este vacia
        printf("\nNo hay nodos\n");
        return -1;
    }

    while (primer_nodo != NULL ){ //Recorre la lista
        if(primer_nodo-
>tamanio_ocupado != 0) { // Contabiliza si un proceso tiene un valor en espacio ocupado
            primer_nodo->contador_proceso++;
            if(primer_nodo-
>contador_proceso > CONTADOR) {// Si un proceso ya lleva mas de 5 unidades de tiempo se reinicia
                primer_nodo->tamanio_ocupado = 0; // El tamanio ocupado se iniicaliza en 0
                primer_nodo->tamanio_desocupado = primer_nodo-
>tamanio_particion; // El tamanio desocupado se reinicia al valor que tenia antes
                primer_nodo->contador_proceso = 0;  // Contador se inicializa en 0
            }
        } else {
            if (primer_nodo-
>tamanio_desocupado >= tamanio_particion) { // Verifica para poder insertar el proceso en el peor bl
oque de memoria
                /*
                Comience eligiendo cada proceso y encuentre el tamanio de bloque maximo
                que se puede asignar al proceso por insertar
                */
                if(worst_posicion == -1) {
                    tamanio_nodo = primer_nodo->tamanio_desocupado;
                    worst_posicion = primer_nodo->posicion;
                } else {
                    /*
                    Si no, deja ese proceso y sigue revisando
                    los procesos posteriores.
                    */
                    if (tamanio_nodo < primer_nodo->tamanio_desocupado) {
                        worst_posicion = primer_nodo->posicion;
                        tamanio_nodo = primer_nodo->tamanio_desocupado;
                    }
                }
            }
```

```
        }
        primer_nodo = primer_nodo->siguiente_nodo;
    }
    return worst_posicion; // Se regresa la peor posicion para insertar
}
```

1. Declare dos variables de tipo entero:
   - La variable **worst_posicion** se va a encargar de encontrar el bloque que peor se ajuste para el proceso insertado.
   - La variable **tamanio_nodo** es para asignarle el **tamanio_desocupado** que tiene la memoria e ir comparando si le conviene insertar al proceso.
2. Al igual que **First-Fit** el **while** es para ir recorriendo toda la lista, dentro de ese hay:
   - Un **if(primer_nodo->tamanio_ocupado != 0)** que es para verificar si un proceso ya está en memoria, y empezar a contabilizar las 5 unidades de tiempo que debe de estar ahí. Después de eso se empieza a aumentar el contador en 1.
   - El segundo **if(primer_nodo->contador_proceso > CONTADOR)** anidado es para verificar si el **tamanio_particion** lleva más de 5 unidades de tiempo en la memoria.
   - Si no se cumple la primera condición del **if** en donde se compara si el **tamanio_ocupado** es diferente de 0, entonces hay 3 **if's** después**.**
   - El primer **if(primer_nodo>tamanio_desocupado>=tamanio_particion)** es para esta insertando en la lista el proceso.
   - El segundo **if(best_posicion == -1)** comienza a elegir cada proceso que encuentre el tamaño del bloque mínimo de memoria que se puede asignar al proceso por insertar.
     - **if(tamanio_nodo < primer_nodo->tamanio_desocupado)** Si no, deja ese proceso y sigue revisando los procesos posteriores.
   - Se regresa la peor posición que encontró el proceso.

```
/*
Este metodo compara el tamaño de memoria requerido por el proceso con los agujeros libres
en la lista. Le asigna al proceso el agujero más grande.
*/
void worst_fit(struct Nodo *primer_nodo) {
    int i, tamanio_particion;
    int worst_posicion = -1;

    if(esta_vacia(primer_nodo)){ // Verifica que la lista este vacia
        printf("\nLa lista esta vacia\n");
        return;
    }
```

```
    for (i = 0; i < 100; i++){ // Se le asigna 100 veces una particion a la lista
        tamanio_particion = generador_random(3, 10);  // Genera los numeros aleatorios entre 3 y 10
        worst_posicion = buscar_worst_nodo(primer_nodo, tamanio_particion); // Empieza la busqueda p
ara inserta el proceso
        if( worst_posicion != -
1) { // Si se puede encontrar un bloque de memoria para el proceso insertado
            while (primer_nodo != NULL){ // Empieza a recorrer la lista
                if(worst_posicion == primer_nodo-
>posicion) { // Si el la posicion del bloque de memoria es igual a la posicion del nodo
                    primer_nodo->tamanio_desocupado -
= tamanio_particion; // Se suma el valor insertado en el espacio ocupado
                    primer_nodo-
>tamanio_ocupado += tamanio_particion; // Resta el valor que tiene el tamnio desocupador menos el va
lor insertado
                    break;
                }
                primer_nodo = primer_nodo->siguiente_nodo; // Avanza al siguiente nodo
            }
        }
    }
}
```

Luego de hacer el método de **buscar_worst_nodo** implemente el método para **worst_fit.**

6. Declare tres variables de tipo entero:
    - La variable **tamanio_particion** es la que va a estar insertando el valor en la lista.
    - La variable **i** va a llevar el conteo.
    - La variable **best_posicion** es para asignarle el método **buscar_worst_nodo**.
7. Se tiene el primer **if** que solo verifica si la lista no está vacía.
8. El **for** es para estar insertando 100 veces los valores a la lista y no estar insertando uno por uno. Dentro del **for** empiezo a generar los números aleatorios y le asigno a la variable **worst_posicion** el método de **buscar_worst_nodo** para que empiece la búsqueda.
9. Después del **for** hay un **if(worst_posicion != 1)** es para ver si puede encontrar un bloque de memoria para el proceso insertado.
10. El **while** es para estar recorriendo la lista, dentro de eso hay:
    - Un **if(worst_posicion == primer_nodo->posicion)** verifica que la posición que encontró el proceso sea igual a la del nodo, para poder insertar el proceso.
    - Después se le suma el valor insertado al **tamanio_ocupado** y después se le resta al **tamanio_desocupado** menos el valor insertado.

Implementación para Blocks Splitting.

Para la implementación de **Blocks Splitting** se me dificulto al momento de dividir el bloque, estuve investigando como lo podía hacer, pero no me funciono. Hice varios métodos, pero tampoco me funcionaron.

Había hecho un método llamado **Split,** pero lo único que hacía era dividir la lista y eso que estaba haciendo es incorrecto. A continuación, lo muestro:

```
void split(struct Nodo *primer_nodo){
        int i=0, tamanio_particion;

        // Hacer que el puntero p1 apunte al primer nodo
        struct Nodo *p1= primer_nodo;
        for(i = 0; i < 100; i++) {
            tamanio_particion = generador_random(3, 10);
            while(primer_nodo != NULL){
                if(primer_nodo-
>tamanio_ocupado != 0) { // Contabiliza si un proceso tiene un valor en espacio ocupado
                    primer_nodo->contador_proceso ++; // Se aumenta el contador
                    if(primer_nodo-
>contador_proceso  > CONTADOR){ // Si un proceso ya lleva mas de 5 unidades de tiempo se reinicia
                        primer_nodo->tamanio_ocupado = 0; // El tamanio ocupado se iniicaliza en 0
                        primer_nodo->tamanio_desocupado = primer_nodo-
>tamanio_particion; // El tamanio desocupado se reinicia al valor que tenia antes
                        primer_nodo->contador_proceso = 0; // Contador se inicializa en 0
                    }
                } else {
                    if(tamanio_particion < primer_nodo-
>tamanio_desocupado/tamanio_particion){
                        p1->tamanio_desocupado -
= tamanio_particion; // Se suma el valor insertado en el espacio ocupado
                        p1->tamanio_ocupado += tamanio_particion;
                    }
                }
                primer_nodo = primer_nodo->siguiente_nodo;
            }
        }

        p1->siguiente_nodo = NULL;
}
```

Entonces lo único que hice, pero no sé si esta bien, al momento de encontrar el valor mas grande como **Worst-Fit** fue dividir el **tamanio_desocupado** entre el **tamanio_particion**.

```
if (tamanio_nodo < primer_nodo-
>tamanio_desocupado/tamanio_particion ) { // Divido el tamanio_desocupado para q
ue proceso pueda ser insertado
                        worst_posicion = primer_nodo->posicion;
                        tamanio_nodo = primer_nodo->tamanio_desocupado;
                    }
```

Fue el único algoritmo que tuve más problemas a comparación de los anteriores.

Observaciones comparando los algoritmos.

First-Fit

Consiste en asignar la primera partición o agujero libre lo suficientemente grande como para acomodar el proceso. Termina después de encontrar la primera partición libre adecuada.

Ventaja:

El algoritmo más rápido porque busca lo menos posible.

Desventaja:

Las áreas de memoria restantes no utilizadas que quedan después de la asignación se desperdician si son demasiado pequeñas. Por lo tanto, no se puede realizar la solicitud de mayor requisito de memoria.

Best-Fit

El **Best-Fit** se ocupa de asignar la partición libre más pequeña que cumpla con el requisito del proceso solicitante. Este algoritmo busca primero en toda la lista de particiones libres y considera el hueco más pequeño que sea adecuado. A continuación, intenta encontrar un hueco que se acerque al tamaño real del proceso necesario.

Ventaja:

La utilización de la memoria es mucho mejor que el primer ajuste, ya que busca primero la partición libre más pequeña disponible.

Desventaja:

Es más lento e incluso puede tender a llenar la memoria con pequeños agujeros innecesarios.

Worst-Fit

El enfoque de **Worst-Fit** consiste en localizar la mayor partición libre disponible para que el proceso que quede sea lo suficientemente grande como para ser útil. Es lo contrario del **Best-Fit**.

Ventaja:
Reduce la tasa de producción de pequeños huecos.

Desventaja:
Si un proceso que requiere mayor memoria llega en una etapa posterior, entonces no se puede acomodar porque el agujero más grande ya está dividido y ocupado.

Blocks splitting

La idea es dividir un agujero más grande, tomando de este un el pedazo más pequeño requerido, mientras que la otra parte quedará libre y puede ser utilizada en otras solicitudes de asignación.
Ventaja:
Se divide el bloque de memoria mas grande para que otros procesos puedan ser insertados ahí mismos.

Desventaja:
Puede pasar que no haya un bloque mayor al proceso y entonces no se pueda dividir.

A continuación, muestro de donde me base para hacer el algoritmo de **First-Fit, Best-Fit, Worst-Fit y Block Splitting.**

Referencias:

1. https://www.geeksforgeeks.org/program-best-fit-algorithm-memory-management/
2. https://www.youtube.com/watch?v=VGt1asAexEM
3. https://www.geeksforgeeks.org/buddy-memory-allocation-program-set-1-allocation/
4. https://www.cs.au.dk/~gerth/papers/actainformatica05.pdf